Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 4: this Friday @ 11:59 pm
  - Lab 3: next Monday @ 11:59 pm
  - Assignment 1: Monday Oct 10^{th} @ 11:59 pm
- **Live support session** for Assignment 1
  - Over Zoom this Friday @ 5:00 pm
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous lecture

- R-way Radix Search Tries
- De La Briandais (DLB) Tries

This Lecture

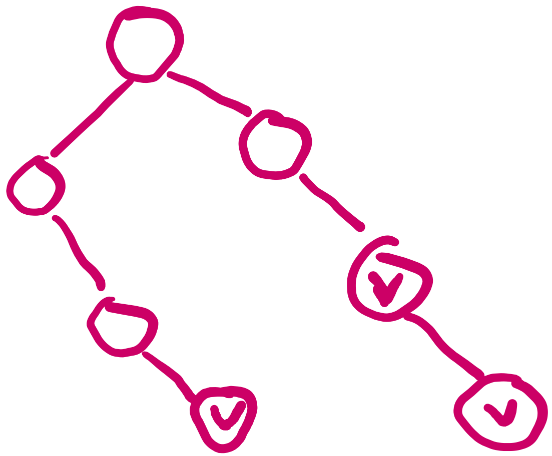
- Compression

Muddiest Points

- **Q: When creating a DST, does it have to start handling keys starting with the leftmost bit, or can it also handle them by starting with the rightmost bit?**
- The algorithm can go either way on the bitstring of the key as long as the direction is the same for all operations

Muddiest Points

- **Q: Would a trie be able to contain a value with less bits than the root, and if so how?**
- In a trie, none of the nodes (including the root) contains any key
- If the questions is “can a trie contain keys of different bit lengths?”,
  - the answer is yes
  - Interior nodes have non-null values in that case
  - The trie here has three keys
    - 011
    - 111
    - 11



Muddiest Points

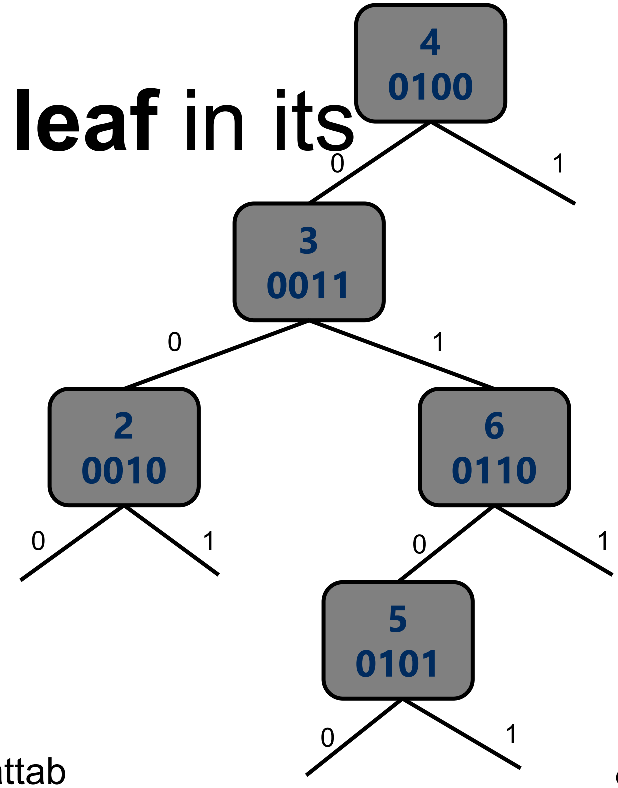
- **Q: How is a Trie different from a Red-Black BST?**
- a trie is different from a search tree because trie doesn’t store the keys inside its nodes but a search tree does
- More in the next question

Muddiest Points

- **Q: when would you use a DST or an RST?**
- **Q: What's the application of RST?**
- DST and RST are efficient in checking if a target key is a prefix of any of the keys in the tree
  - e.g., making routing decisions in the Internet
- DSTs are preferred over BSTs when bits of keys are randomly distributed (i.e., the probability of each bit being zero is 0.5)
  - The DST will be balanced in this case without having to use the more complicated Red-Black BST
- RSTs are preferred over BSTs when bit lengths of keys are close to *log n*
  - The RST will be balanced in this case without having to use the more complicated Red-Black BST
- Note that DST and RST don't provide the extra operations (e.g., predecessor and successor) provided by BST

Muddiest Points

- **Q: how can any node in 3's subtree replace 3 in DST example**
- Because all nodes in 3’s subtree share a common prefix with length 1 with 3
  - The node that replaces 3 will still be found using the DST search algorithm
- For simplicity, we replace 3 with any **leaf** in its subtree



Muddiest Points

- **Q: Could you spend more time going through new lecture**
- Sometimes, addressing the muddiest points takes up a large portion of class time
- Usually, new material is embedded between the muddiest points responses

Muddiest Points

- **Q: Is the insertion position for DST based on the first bit that is different from the last insert? Or is it based on the relative comparison to last insert?**
- DST Add Algorithm for adding a key *k* and a corresponding value
  - if root is null, add *k* at the root and return
  - current ← root
  - if k is equal to the current node's key, replace value and return
  - if current bit of k is 0,
    - if left child is null, add *k* as left child
    - else continue to left child
  - if current bit of k is 1,
    - if left child is null, add *k* as right child
    - else continue to right child

Muddiest Points

- **Q: When is DST preferable to radix search trie?**
- A: When bit lengths of keys are >> log n

Muddiest Points

- **Q: I don't understand the advantage of making another node in the DLB instead of the tree structure.**
- A: DLB saves space when the number of children per node in an R-way RST is small

Muddiest Points

- **Q: How does DLB save space over r way trie? Example please?**
- Let's the set of keys:
  - ksm1 … ksm9
- How big does an 256-way RST take vs. a DLB trie?

R-way RST

```
private class Node {
    private Object val;
    private Node[] next;

     private Node(){

        next = new Node[R];

     }
}
```

Each node takes 4*(R+1) = 4*257 = 1028 bytes, assuming 4 bytes per reference variable

R-way RST

Val: null
Next
k
Val: null
Next
s
Val: null
Next
m
Val: null
Next
1 2 3 4 5 6 7 8 9
Val: V_{ksm1}
Next
Val: V_{ksm2}
Next

R-way RST

We will end up with 4 + 9 = 13 nodes

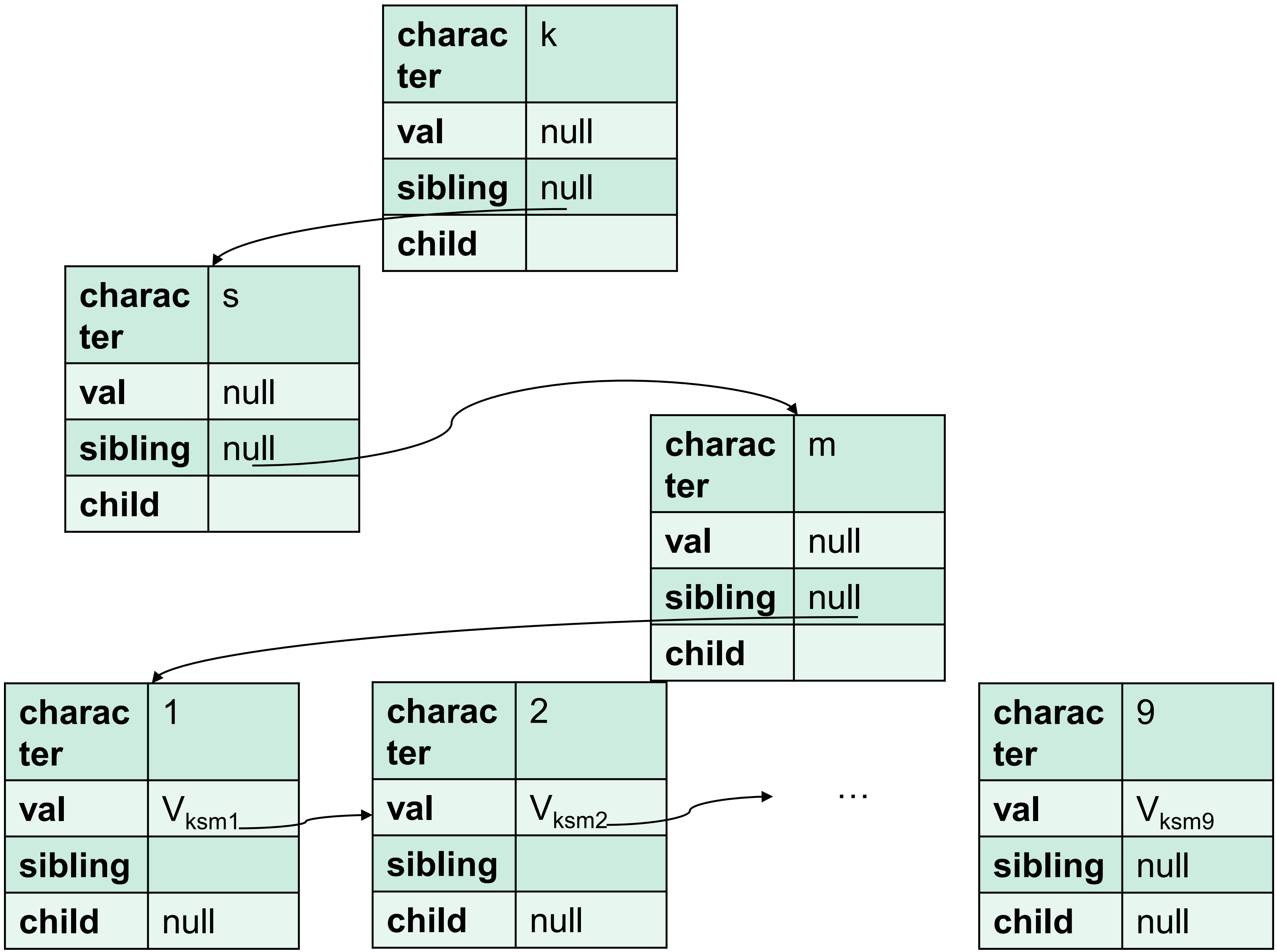
Total space is 13*1028 = 13,364 bytes

DLB Trie

```
private class DLBNode<T> {
    private Character character;
    private Object val;
    private Node sibling;
    private Node child;
}
```

Each node takes 4*4 = 16 bytes, assuming 4 bytes per reference variable

DLB Trie



character k
val null
sibling null
child
character s
val null
sibling null
child
character m
val null
sibling null
child
character 1
val V_{ksm1}
sibling
child null
character 2
val V_{ksm2}
sibling
child null
…
character 9
val V_{ksm9}
sibling null
child null

DLB Trie

We will end up with 12 nodes

Total space is 12*16 = 192 bytes

Compare to 13,364 bytes with an R-way RST

Muddiest Points

- **Q: What determines the number of bits you use for the bit representation of a key in DSTs and RSTs?**
- Typically, the number of bits is determined by the application
  - e.g., keys are Pitt usernames, PeopleSoft IDs, English sentences, etc.
- It is better to re-encode the keys to have a bit length of *log n* bits each
  - Requires extra space to store the mapping from old keys to new keys
  - sometimes not possible: e.g., when *n* is not known in advance
- Better yet, we can assign bit lengths based on frequency of access:
  - Shorter bitstrings for more frequently accessed keys
    - Results in smaller ***average*** search time

$$\text{average case runtime} = \sum_{\text{all cases}} Pr(\text{Case}_i) \times \text{runtime for Case}_i$$

Muddiest Points

- **Q: Not really a muddiest point, but it would be extremely helpful to see actual code (not pseudo code) next to some of these trees**
- You will see code in the recitations

Muddiest Points

- **Q: DLB do what?**
- De La Briandais (DLB) Trie
  - tree-like structure used for searching when keys are sequences of characters
  - each nodelet
    - stores one character,
    - points to a sibling (linked list of siblings), and
    - points to a child
  - worst-case running time is O(wR)
    - w: number of characters in the key
    - R: alphabet size
  - worst-case can be avoided by using DLB only when the sibling lists are short
  - check add algorithm in previous lecture

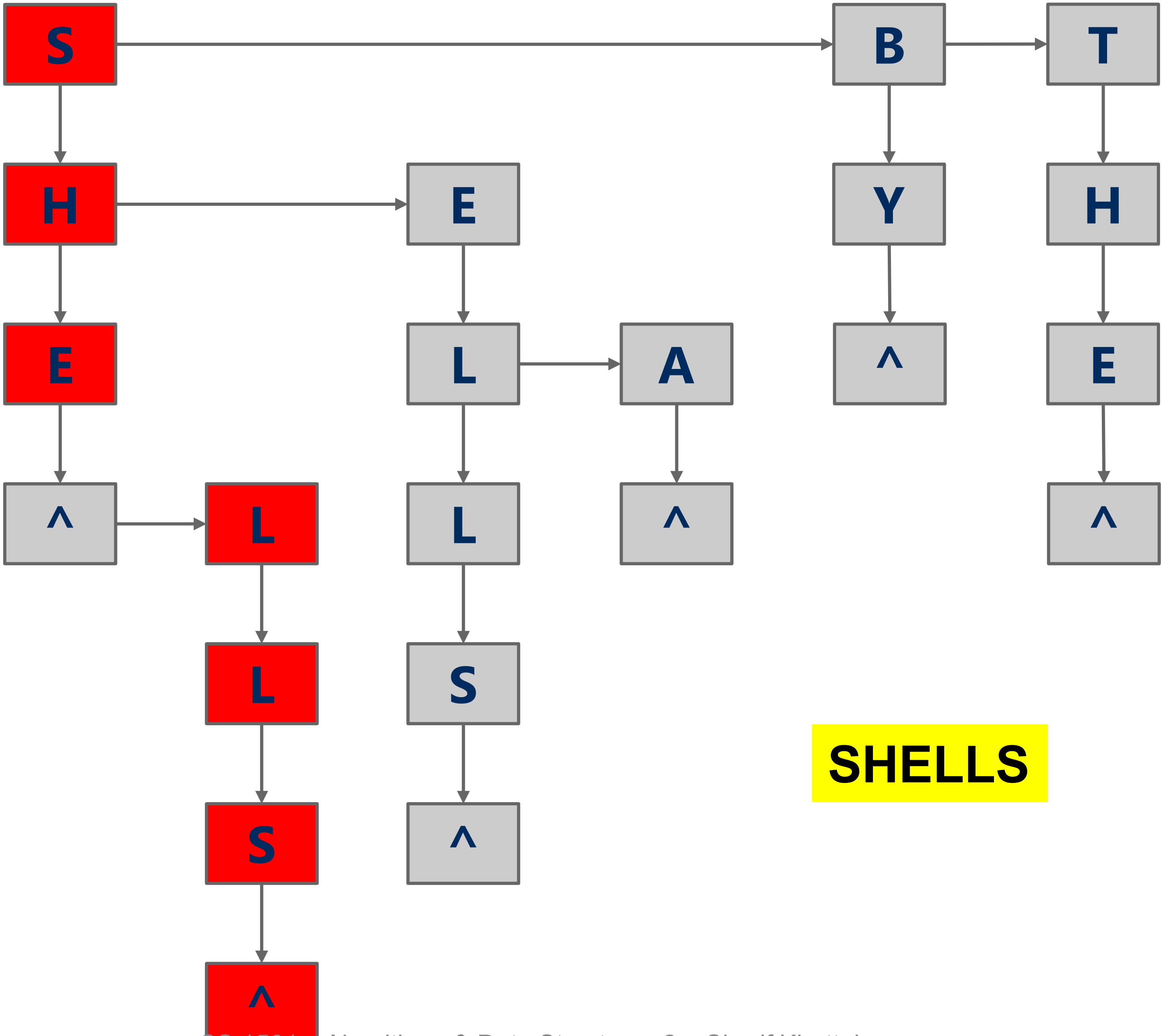
Muddiest Points

- **Q: Just wanted to quickly double check some details about DSTs: The max height of a DST is the number of key bits + 1**
- Correct
- **Q: and the max comparisons you can do is the key's bit length, correct?**
- No, it is also b + 1

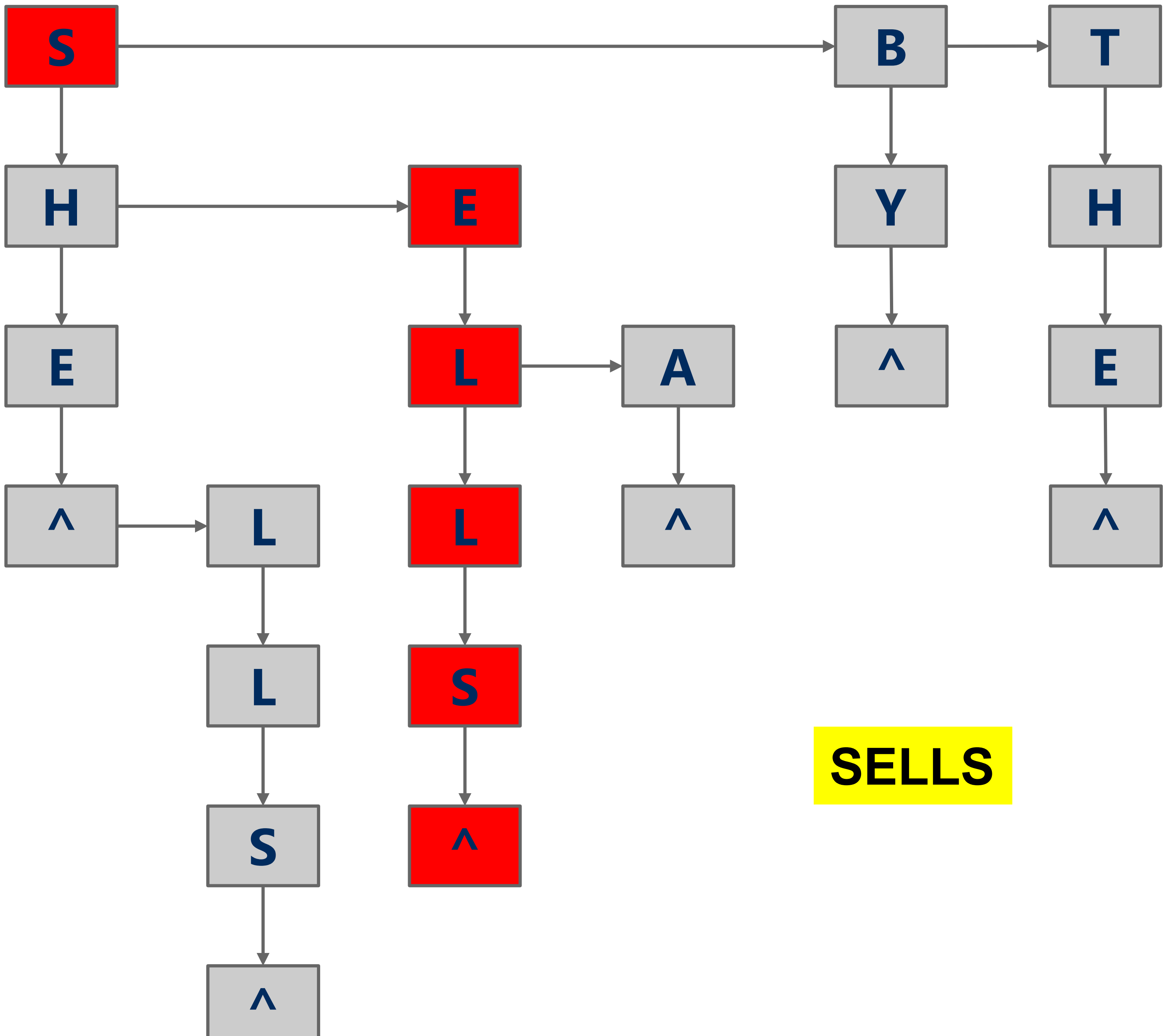
Muddiest Points

- **Q: why was there two paths for "shell" on the DLB example?**
- There was only one path!

DLB Example



DLB Example



Muddiest Points

- **Q: What is the point of the sentinel character? How does that implementation of the DLB differentiate it from the other DLB implementation?**
- If the DLB stores only keys (without corresponding values), we don’t need the val field in the DLBNode
- But, val helped us determine if the node we stop at corresponds to a key or not
  - when val is not null, the node corresponds to a key
- Without val, we need a different method: using a sentinel
  - the sentinel is added to each key before adding and before searching
  - a key is found when the key with the sentinel is found
  - e.g., adding she results in adding she^
  - searching for she becomes searching for she^
  - if “she^” is found then she is a key
  - if only “she” is found (without the sentinel), she is not a key

Muddiest Points

- **Q: what sorting methods have the data explicitly and when is it implicit.**
- Tries store keys implicitly, whereas trees store keys explicitly inside tree nodes
- Check node structure for a tree vs. a trie

Muddiest Points

- **Q: Can you re-explain what you meant by w = b/ceiling(logR)?**
- The string "she" has 3 characters (*w=3)*
- If we look at the bit representation of "she", assuming each character is an extended ASCII character (i.e., 8-bit character), the number of bits will be $b = 3*8 = 24$
- For extended ASCII, the alphabet size is $R = 2^8 = 256$
- b = b/8 = b/logR

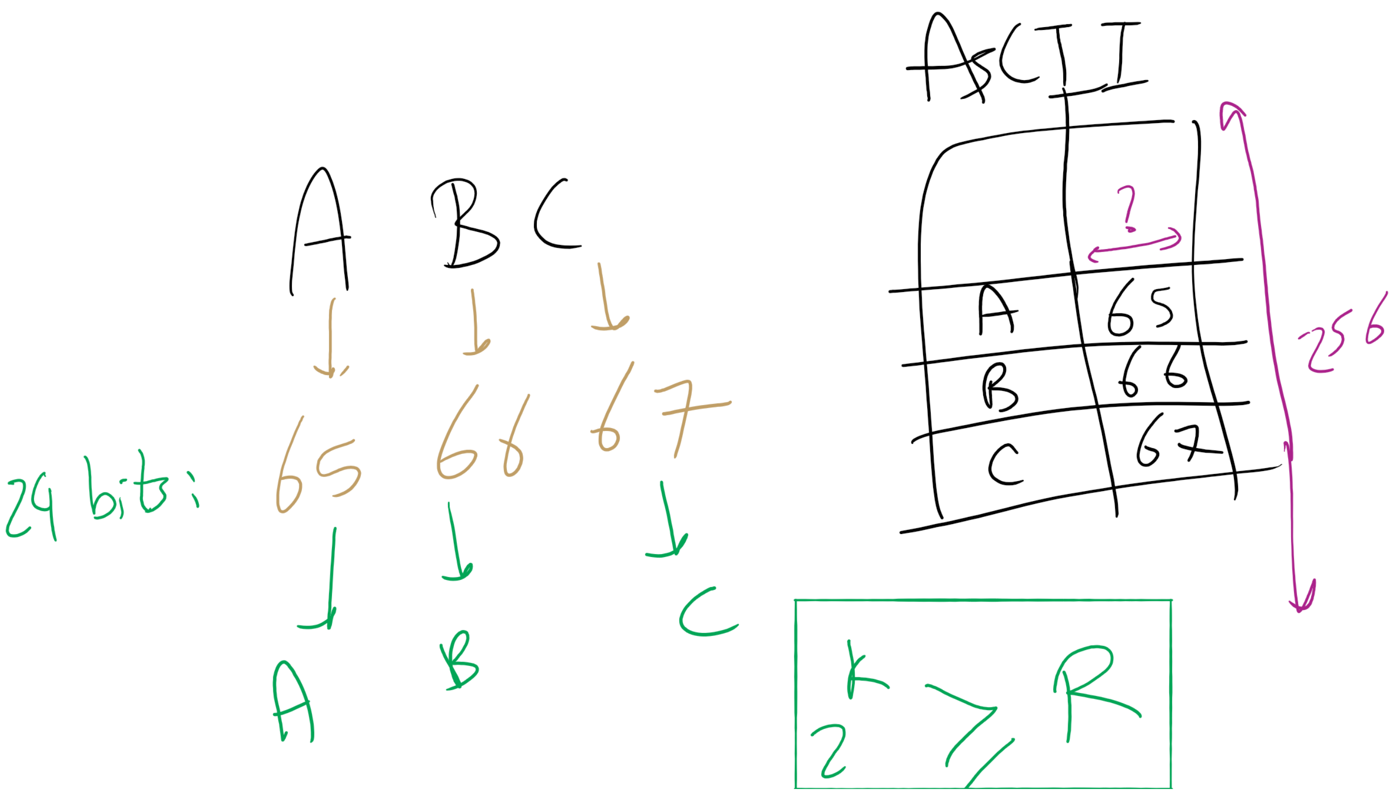
Problem of the Day: Compression

- Input: A file containing a sequence of characters
  - *n* characters
  - each encoded as an 8-bit Extended ASCII
  - total file size = 8*n
- Output: A shorter bitstring
  - of length < 8*n
  - such that the original sequence can be fully restored from the bitstring

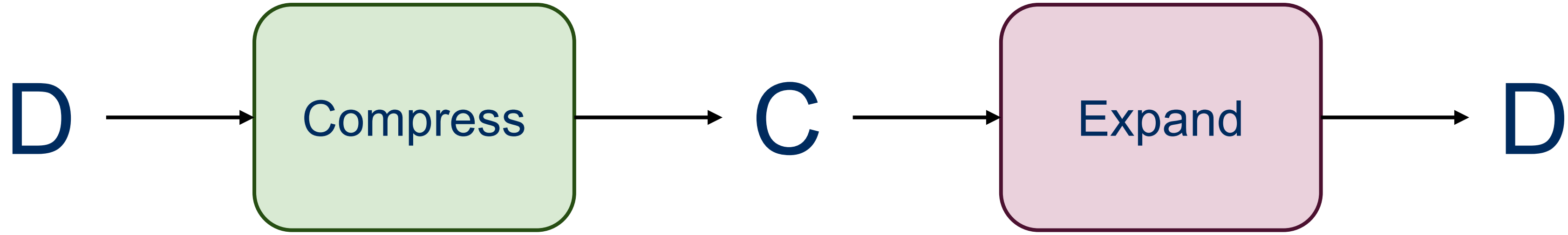
ASCII Encoding

A set of R symbols can be represented using fixed-size encoding of length *k* bits each

- iff $2^k >= R$
- that is, $k = \lceil \log R \rceil$



Lossless Compression

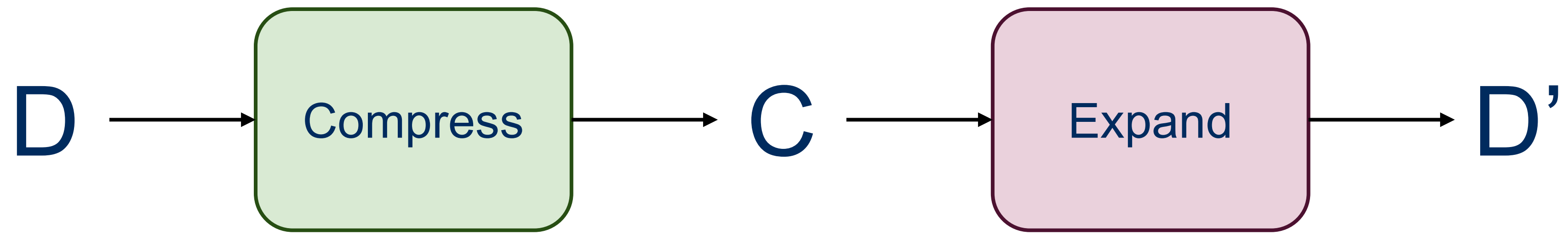


- Input can be recovered from compressed data exactly

- Examples:

  - zip files, FLAC

Lossy Compression



- Information is permanently lost in the compression process
- Examples:
  - MP3, H264, JPEG
- With audio/video files this typically isn’t a huge problem as human users might not be able to perceive the difference

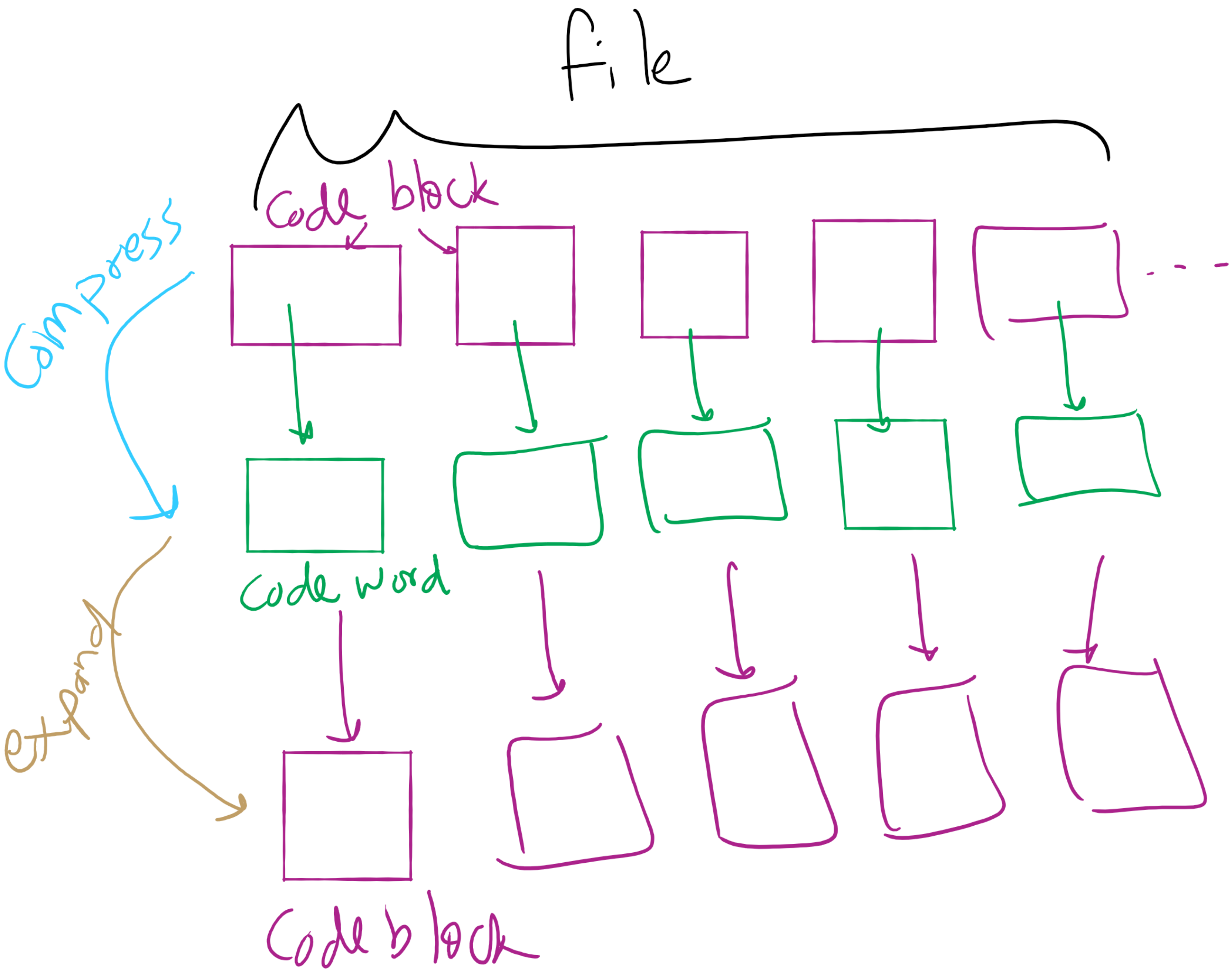
Lossy examples

- MP3
  - "Cuts out" portions of audio that are considered beyond what most people are capable of hearing
- JPEG

40K

28K

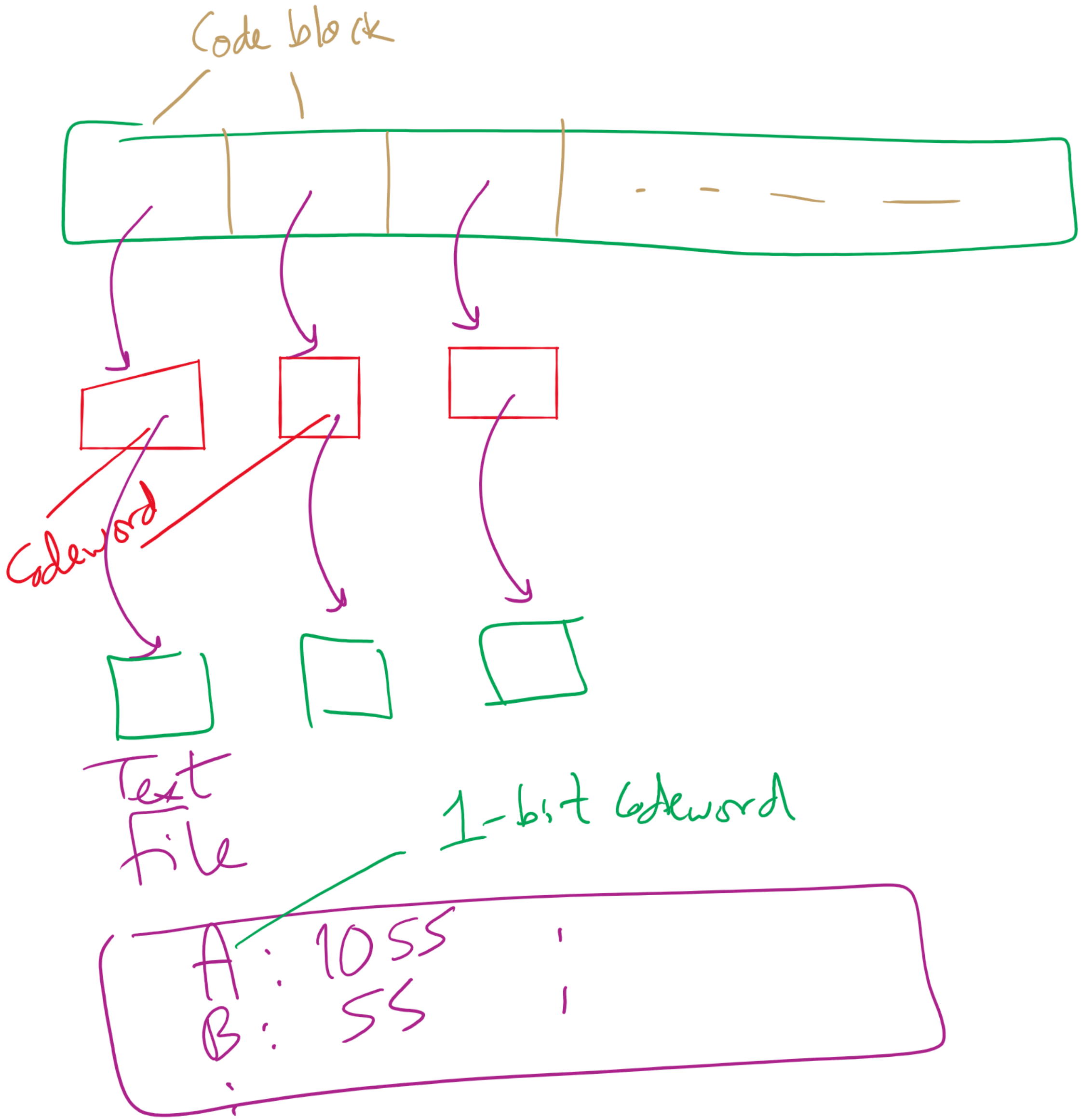
Lossless Compression Framework



Solution 1: Huffman Compression

- What if we used *variable length* codewords instead of the constant 8? Could we store the same info in less space?
  - Different characters are represented using codes of different bit lengths
  - If all characters in the alphabet have the same usage frequency, we can't beat block storage
    - On a character by character basis…
  - What about different usage frequencies between characters?
    - In English, R, S, T, L, N, E are used much more than Q or X

Variable size codewords



But we have to worry about restoring the data!

- Decoding was easy for block codes
  - Grab the next 8 bits in the bitstring
  - How can we decode a bitstring that is made of variable length code words?
  - BAD example of variable length encoding:

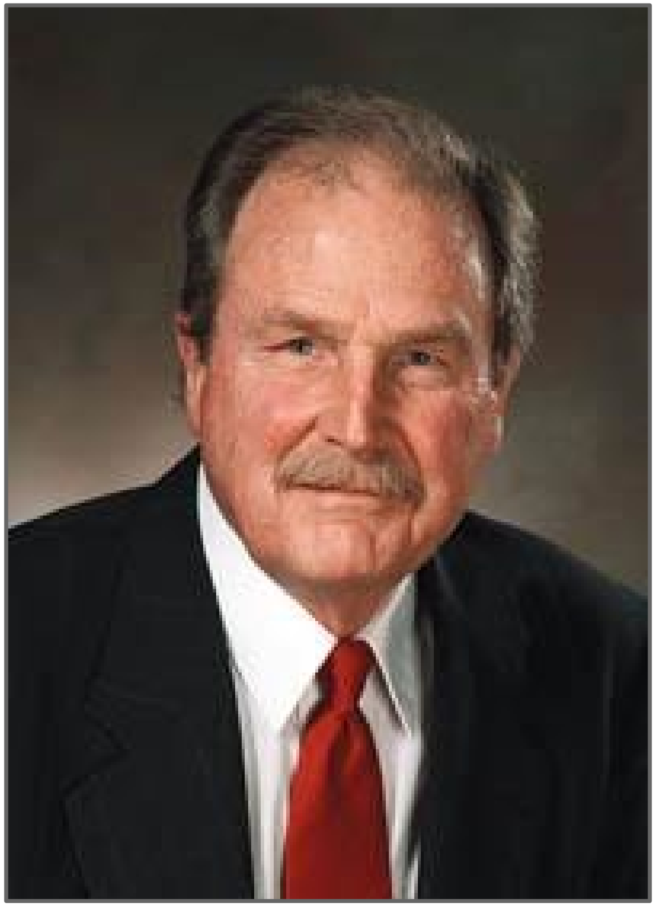
1	A
00	T
01	K
001	U
100	R
101	C
10101	N

Variable length encoding for lossless compression

- Codes must be *prefix free*
  - No code can be a prefix of any other in the scheme
  - Using this, we can achieve compression by:
    - Using fewer bits to represent more common characters
    - Using longer codes to represent less common characters

How can we create these prefix-free codes?

Huffman encoding!



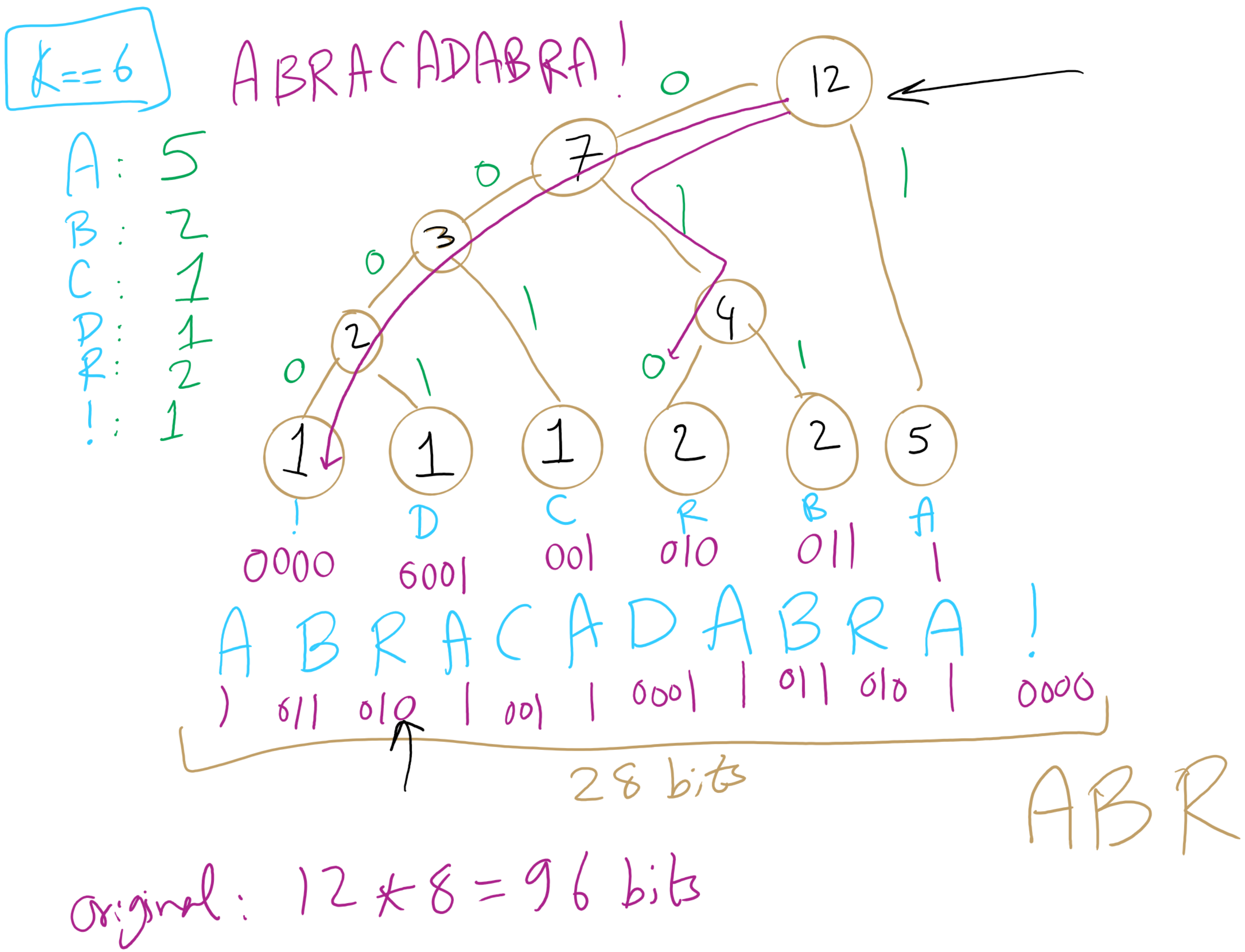
Subproblem: Prefix-free Compression

- Input: A sequence of *n* characters
- Output: A codeword h_i for each character *i* such that
  - No codeword is a prefix of any other
  - When each character in the input sequence is replaced with each codeword
    - the length of that compressed sequence is minimum
    - the original sequence can be fully restored from the compressed bitstring

Generating Huffman codes

- Assume we have K characters that are used in the file to be compressed and each has a weight (its frequency of use)
- Create a forest, F, of K single-node trees, one for each character, with the single node storing that char's weight
- while |F| > 1:
  - Select T1, T2 ∊ F that have the smallest weights in F
  - Create a new tree node N whose weight is the sum of T1 and T2's weights
  - Add T1 and T2 as children (subtrees) of N
  - Remove T1 and T2 from F
  - Add the new tree rooted by N to F
- Build a tree for "ABRACADABRA!"

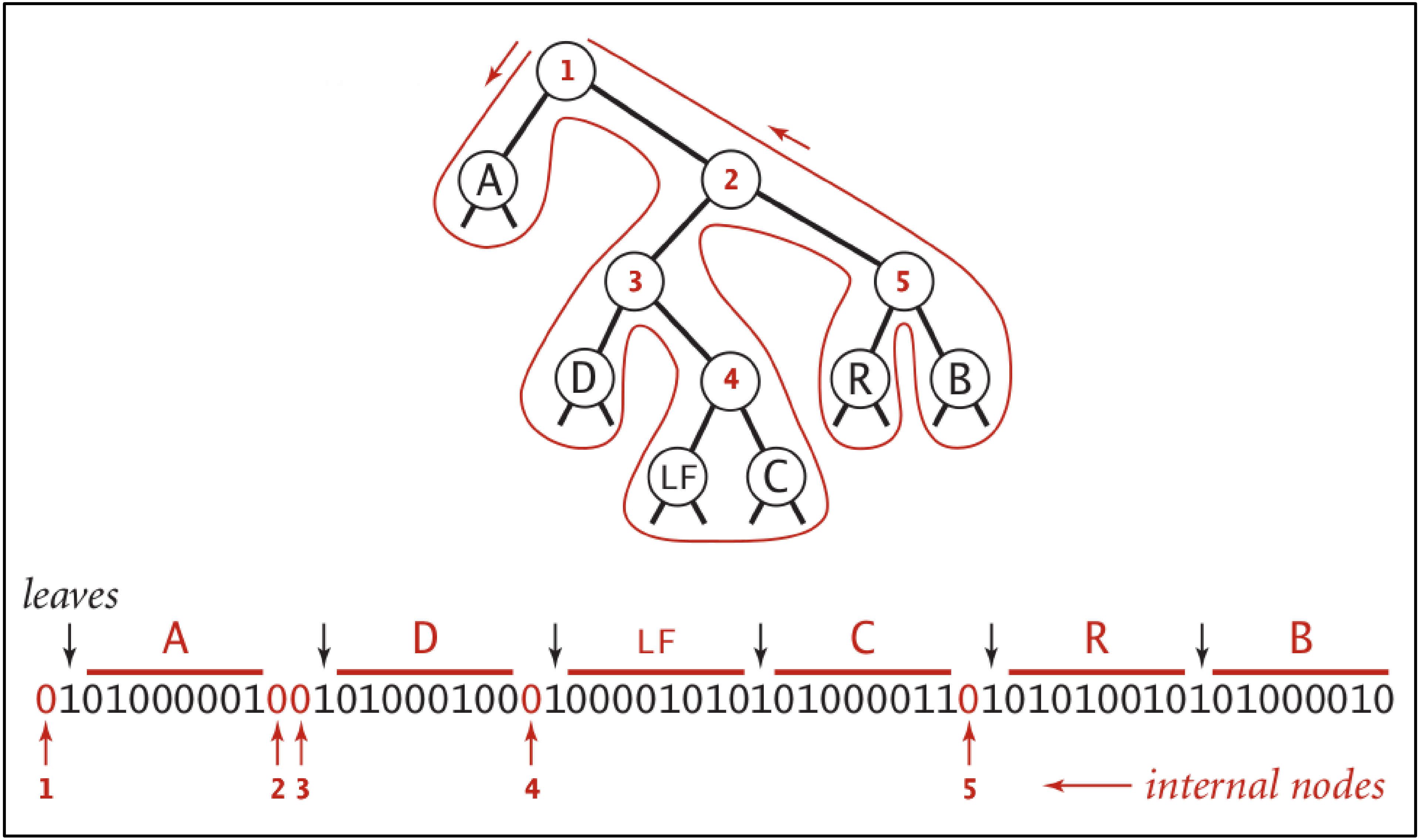
Huffman Tree Construction Example



Implementation concerns

- To encode/decode, we'll need to read in characters and output codes/read in codes and output characters
  - …
  - Sounds like we'll need a symbol table!
    - What implementation would be best?
      - Same for encoding and decoding?
  - Note that this means we need access to the trie to expand a compressed file!

Representing tries as bitstrings



Binary I/O

```
private static void writeTrie(Node x){

    if (x.isLeaf()) {

        BinaryStdOut.write(true);

        BinaryStdOut.write(x.ch);

        return;
    }
    BinaryStdOut.write(false);

    writeTrie(x.left);

    writeTrie(x.right);
}

private static Node readTrie() {

    if (BinaryStdIn.readBoolean())

        return new Node(BinaryStdIn.readChar(), 0, null, null);

    return new Node('\0', 0, readTrie(), readTrie());

}
```

Binary I/O

```
private static void writeBit(boolean bit) {
        // add bit to buffer
        buffer <<= 1;
        if (bit) buffer |= 1;
        // if buffer is full (8 bits), write out as a single byte
        N++;
        if (N == 8) clearBuffer();
}
```

```
writeBit(true);
writeBit(false);
writeBit(true);
writeBit(false);
writeBit(false);
writeBit(false);
writeBit(false);
writeBit(true);
```

buffer: 00000000

N: 0